A Language for Describing Agent Behavior as Hybrid Models

Suresh K. Kannan*, Claus Christmann†,
Seungman Lee‡, Eric N. Johnson§, So Y. Kim¶
270 Ferst Drive, School of Aerospace Engineering
Georgia Institute of Technology, Atlanta, GA 30332

Air Traffic conflict detection and resolution (CDR) involves multiple domains, the modeling of physical systems such as aircraft, encoding conflict detection algorithms as well as the procedures(tasks) for conflict resolution. Depending on the analysis being conducted, an implementation language is usually chosen to cater for easy rendering of algorithms in the primary domain of interest. The more specialized the choice of implementation language, the greater the difficulty in expanding the fidelity of models in other domains. This paper takes a unified view of continuous equations, algorithms and procedures. Events that occur in sequence as well as in parallel are represented in a unified manner by interpreting them as hierarchical state-charts at a low-level and as procedures or task trees at a higher level. The relationship between the two levels are recognized and utilized in decomposing task trees in to hierarchical state charts and eventually into C++ code for implementation.

I. Introduction

In Air Traffic Management research there exists a multitude of conflict detection and resolution methods (surveyed in[1]), each with its own specific modeling method. A common trait in most of these systems is that the various agents in the system exhibit hybrid behavior, continuous dynamics due to the physical systems such as aircraft dynamics, and discrete modes of operation such as the modes of the flight management system (FMS). In this paper we propose a language that enables the intuitive description of agent behavior when simulating it's physical dynamics as well as other it's aspects, such as, control and behavior algorithms. Throughout this paper, *agents* denote multiple aircraft, the air traffic controller, radar, pilots and any other object that may be relevant to the scenario being modeled. *Procedure* implies a list of actions or tasks that must be executed in order to accomplish a *goal*. *Procedural* is used in reference to sequentially executing statements as in C, Fortran and Matlab which are procedural languages. Both terms are similar but with different meaning, which, should be clear when used in context. Additionally, *task* refers to instances of *procedures* and an agents response to a *command*, *procedure*, *occurrence of an event*, *function call* all denote the same construct.

There are various types of languages, that may be used for implementation purposes. Procedural languages such as C and Fortran are appropriate when describing continuous state equations and algorithms for detecting conflicts. On the other hand, the more declarative and goal-directed languages such as Prolog are well suited to describing procedures that represent high-level agent behavior. Indeed, once a set of simple procedures are coded, solutions to simple or complex problems that were not explicitly considered

*Research Engineer, suresh.kannan@ae.gatech.edu
†Research Assistant, hcc@gatech.edu
‡Research Engineer, seungman.lee@gatech.edu
§Assistant Prof., eric.johnson@ae.gatech.edu
¶Research Assistant, soyoung.kim@gatech.edu

may be found (if one exists) by interrogating the Prolog inference engine. Depending on the objective, an appropriate language may be chosen that allows algorithms and procedures to be encoded in a succinct but unambiguous manner. In the current scenario of Air Traffic Management the goal is to evaluate procedures that are fixed and simple enough for human operators to implement with confidence. One of the goals is to test such procedures in a complex setting when communication delays, drop outs and other artifacts are introduced.

Although the primary purpose is the evaluation of procedures, the various agents in the system require vast amounts of procedural programming to represent their behavior with sufficient fidelity. One approach taken in[2] is to code an agent's continuous dynamics, conflict detection algorithms and other procedural functions in C++ for efficient simulation and leave task level logic to a runtime engine that reads in a set of XML files that describes high-level agent behavior. A better approach for the latter purpose is to use one of the task description/decomposition languages[3,4] which are both Lisp based implementations. One could use the universal method of simply linking to C libraries, to gain both procedural and task description facilities. However, the description agent behavior is now split across multiple languages and files. Additionally, the manually-coded C/C++ parts will now also contain various artifacts for communication between Lisp/XML and C. This is undesirable because it pollutes the description of the system.

Our inspiration for the simple language described herein primarily comes from the Lisp based Procedure Definition Langauge.[3] Similar earlier efforts by others are described in work by Simmons on the Task Description Language.[5,6]

The following section on motivation and requirements, describes several observations that helped conclude that almost all constructs required for modeling agents can be represented by hierarchical state machines augmented with continuous dynamics. The section on Language Constructs presents the details of the actual language and also some sample code for an ATC conflict detection and resolution scenario.

II. Motivation and Requirements

A. Continuous Dynamics

All physical systems have continuous dynamics. A relevant requirement is the ability to propagate the finite-dimensional states of every agent. This requirement applies to both the continuous dynamics of agents that represent actual aircraft as well as internal models that may be used by conflict detection modules. For example, an algorithm monitoring two aircraft may use an internal model of the relative dynamics with one of the outputs being the time to minimum separation. The internal models are special because conflict detection methods use varying methods for state propagation. The three most common are *integration*, *worst case* and *probabilistic*. In all three cases, detection algorithms must be able to request fast-time simulations. Hence it should be possible to model equations of the form $\dot{x} = f(t, x, u)$, where, t, can represent varying notions of time.

B. Procedures/Tasks

We wish to embody each agent with a set of procedures that enable itself or other agents to manipulate its configuration. They are a high-level listing of actions and sub-procedures that must occur with a particular, possibly dynamic, ordering to satisfy the top-level goal. Hence,

- Procedures must be decomposable at runtime.
- Statements in procedures may be executed in parallel.
- It should be possible to query the completion status of any sub-procedure.
- Synchronization constructs should be available to hold off on actions or further decomposition based on complex conditions on continuous states, discrete states, the status of sub-procedures and also time periods.

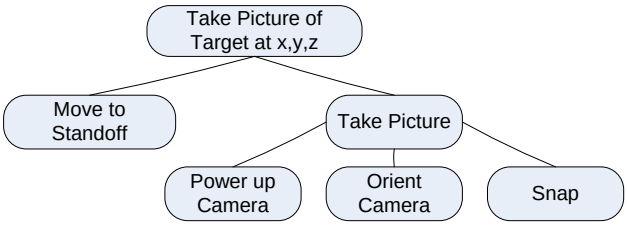

Figure 1: An example of task decomposition (slightly modified example from[3])

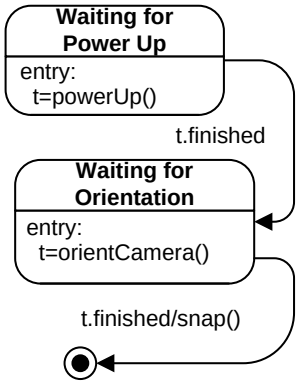

Figure 2: Sequential

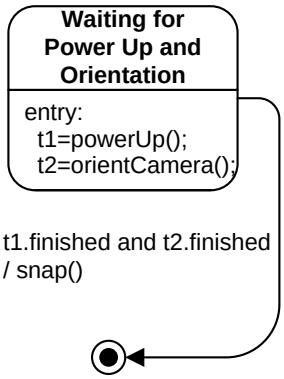

Figure 3: Parallel

For example, the simple task of a UAV moving to a given position and taking a picture may be decomposed into smaller tasks as shown in in Figure 1. Although Figure 1 illustrates the correct decomposition of the task, the execution semantics are incomplete. Assuming parallel execution, behavior is unpredictable and usually incorrect unless additional conditions are imposed. Even though the *Orient Camera* task may not be further decomposed, the camera actuators will take a finite amount of time to move to a new position and is an activity that can take a few seconds. Snapping a picture only makes sense after the orientation task is complete. That condition may be imposed by declaring the constraint, *waitfor(Orient Camera) then Snap*. This is sufficient if we assume that the *Power Up* takes an infinitesimally small time to execute and can be considered to have been completed when the call returns (like a C function call). It is quite possible that *Power Up* itself can take a few seconds to occur, hence the *Snap* action may be synchronized with the completion of its pre-requisites as follows, *waitfor(Power Up) then waitfor(Orient Camera) then Snap* (see Figure 2). A more efficient execution may be specified as *waitfor(Power Up and Orient Camera) then Snap* (see Figure 3). This latter set is more efficient because it allows the power-up and orientation activities to occur at the same time.

With regard to the overall requirement of taking a picture only after reaching the target. Figure 4 illustrates the overall task with the parallel semantics chosen for the *Take Picture* sub-task. If executed as shown in the figure, the *Take Picture* sub-task will not be decomposed until the UAV has reached its target. A more efficient implementation would be replace the `t.finished` condition with something based on range or time to target. If we know a priori that the camera will take approximately 10s to warm up and orient itself, the task decomposition condition may be replaced with `timeToTarget < 10s` and the `snap()` procedure needs an additional precondition that `t.finished` be met in addition to `t1.finished and t2.finished`. This will cause the camera system to be ready just-in-time but a snap will only be taken once the UAV has actually reached its target, powered up and oriented its camera correctly.

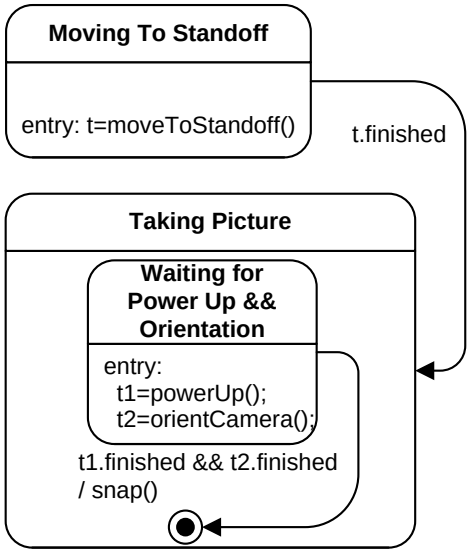

Figure 4: Overall *Move to Target then Take Picture* task

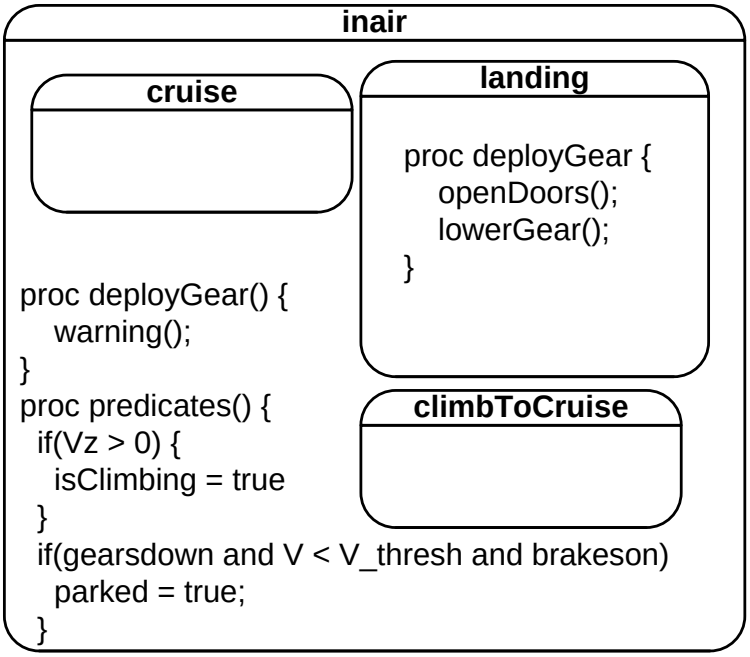

Figure 5: Discrete States

C. Discrete Hierarchical States

Agent behavior can vary depending on its current *mode* or *state*. Both its continuous dynamics as well its response to events (procedures) can depend on the agent's discrete state. A common approach is to model agents as hybrid automata with both continuous and discrete states. Although Finite State Machines can be used to model the simple cases, using Hierarchical State Machines[7] enables a simpler representation of complex behavior.

Suppose an airplane in cruise mode is asked to lower its gear, one response is to simply ignore the event or issue a warning. In Figure 5, the `inair.deployGear` procedure is the default response of the agent when the aircraft is in one of its child states i.e., when in `cruise` or `climbToCruise`, a warning will be issued. During the landing phase, the `deployGear` procedure is a valid command and hence a valid procedure is specified within the `cruise` state. This represents behavior inheritance[8] where events that are not handled explicitly by child states are automatically handled by a matching procedure in the closest parent. If no matches are found, the event is ignored.

D. Predicates

Many times, an agent's hierarchical states do not cater to the view point of an external algorithm. The continuous trajectories of one system may be interpreted as discrete states or *truths* in another. For example, the autopilot may be performing a *coordinated turn* and as long as the aircraft's tracking performance is within limits, a Flight Management System (FMS) that is monitoring the aircraft's continuous states will interpret the aircraft as being *in the coordinated turn* state. The aircraft itself may have no explicit flight mode called *coordinated turn*. Thus the concept of a coordinated turn is purely relevant to the FMS agent and no one else. These *truths* are implemented as *predicates* that are evaluated as being true or false whenever requested and produce no side effects and is the same as checking if a certain set of conditions are true. An example of specifying predicates is shown in Figure 5 where they are specified in the special procedure `predicates`.

E. Other Requirements

- Keep syntax as free as possible from programming language artifacts and allow the most unambiguous, minimal description of the system or equations.
- Keep user free from dealing with memory allocation and deallocation.
- Do not want to deal with pointers and other syntax that can easily introduce bugs.
- Does not have to follow a particular programming paradigm (procedural, functional etc..).
- Standalone. Should not rely on external tools like XML.
- Primary goal is the fast simulation of agents for evaluation and testing of procedures.

III. Language Constructs

The language constructs described here are used to specify files ending with the extension `.r`. The translator that converts these language constructs to C++ is referred to as the *r* compiler . Language details are discussed below using Figure 6 for illustration.

1. Field Declaration

In Figure 6, the partial declaration of two agents is shown, namely an Aircraft and a Flight Management System (FMS). The language has a C like syntax in that it uses curly braces to enclose scopes. The keyword `class` essentially declares an Agent. The basic built-in types are

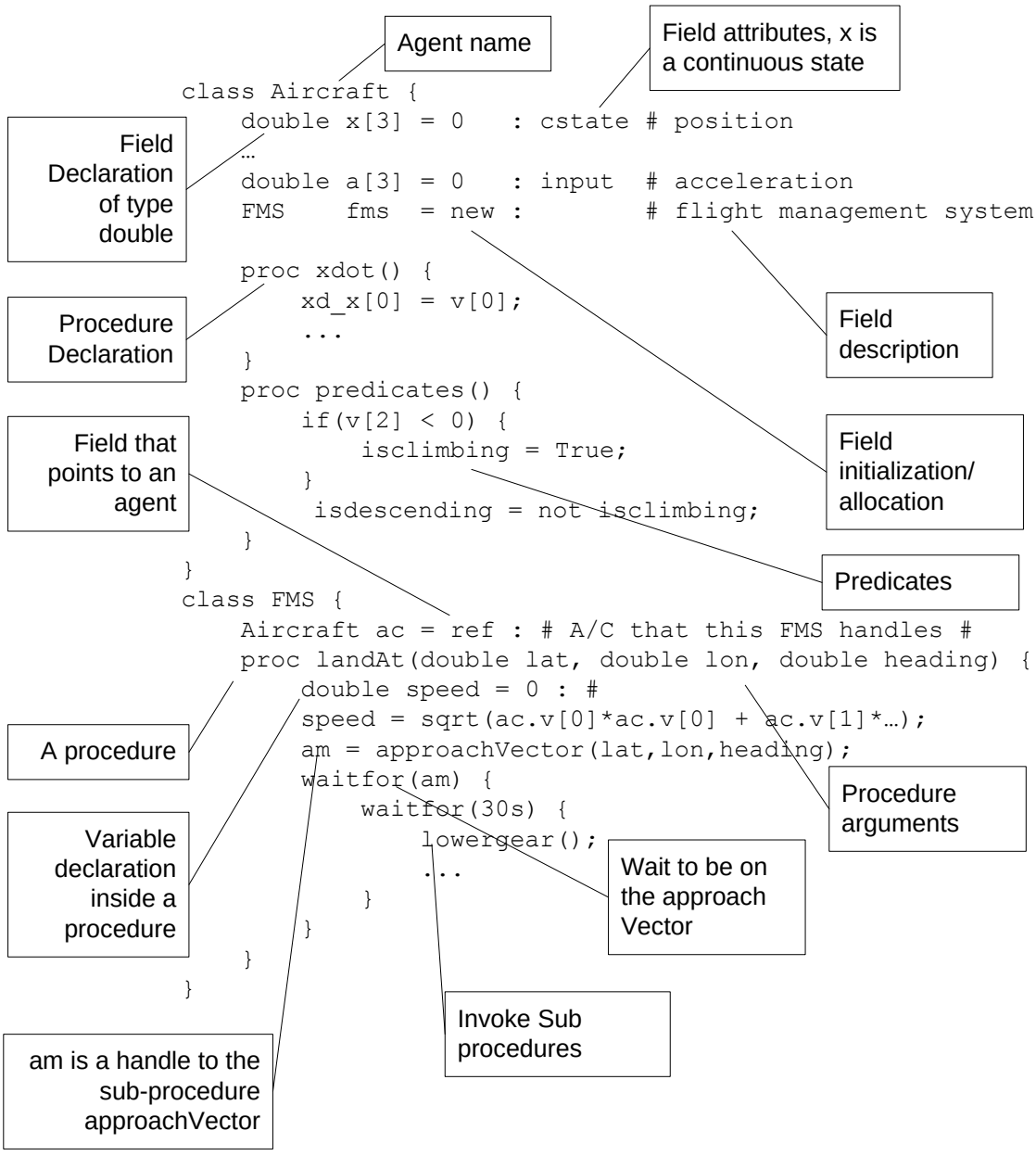

Figure 6: Sample code

`bool , char , int , float , double , string`

Agents have fields that hold the agent's state. Some fields are special and may be labeled as being a part of the continuous state vector or input or output of the agent. Hence in Figure 6, the declaration

```
double x[3] = 0 : cstate # xxxx
```

allocates a one dimensional array of dimension 3, initialized to 0 and will be treated a continuous state. Other fields in the agent may also be labeled as continuous states. Annotation of fields rather than an explicit vector for the state allows equations to be clearer. For example, one does not have to refer to the body roll rate as the fourth element of an aircraft's state vector `x[3]` (first element is `x[0]`). It may be referred as as `w[0]` . If the agent angular rate fields had been specified as scalars, the angular rates would be referred to simply as `p,q,r` in equations or by other agents as `ac.p` where `ac` is a reference to the aircraft object. If a field `x[3]` is declared as a state, the *r* compiler automatically introduces `xd_x[3]` as fields of the agent to hold its derivative. The full set of valid annotations for a field are

`const , input , output , cstate`

Field initialization is of three types, a) `<value>`, b) `new` and c) `ref`. (a) is valid for initializing basic datatypes and implies that it will be allocated. (b) explicitly says the field will be allocated here and (c) indicates the field is simply a pointer and can change over time, no memory is allocated. If the variable is declared as an array, no special notation is necessary when passing the variable around to procedures. An inherent property of every field is its array dimensions.

2. Procedures

In Figure 6, the procedure declaration `xdot` is recognized by the *r* compiler as special and will be called when the continuous state derivative is required. Within the body of a procedure, an agent's own fields may be referenced without any prefixes. The procedure declaration `predicates` is also recognized as special. It essentially serves as a poor-man's state machine, whenever the agent's discrete state configuration is fully defined by the agent's field values (i.e., static). Complex predicates may involve the states of other agents as well, albeit still static. It serve's its purpose well when during the initial coding of an agent the full state machine topology is unknown or just not yet fleshed out. Subsequently, predicates can be used to represent complex conditions which do not necessarily fall within the hierarchical states of the agent. All procedures may be overridden by a discrete state's children states, including the special procedures `xdot` and `predicates`.

3. `waitfor`

All the language features discussed so far fall within within the mainstream constructs of procedural programming languages. If all procedures execute their actions and sub-procedures sequentially then the constructs provided so far suffice. However, the default execution semantics are the parallel execution of procedures. A task synchronization construct is required whenever the execution of a task depends on the result of another or has a set of pre-conditions that have to be met before execution can begin. This is provided by the *waitfor* construct whose syntax is shown below

Listing 1: waitfor Syntax

```
waitfor(<time|task|predicate|state|expression>) {
   <statement>;
   ...
}
```

The `waitfor` can wait on multiple types of conditions all of which may be combined using logical operators. If a task object is waited on, it is equivalent to waiting on the task's finished state i.e, `t1.finished`. A procedure with no `waitfor` constructs can be completed immediately whereas procedures with `waitfor` calls,

```
class UAV {
    bool wow = true : # initially on ground
    proc landAt(double x, double y) {
        t1 = flyTo(x,y);
        waitfor(t1) {
            waitfor(20s) {
                setDescentRate(Vz);
                waitfor(wow);
            }
        }
    }
}
```

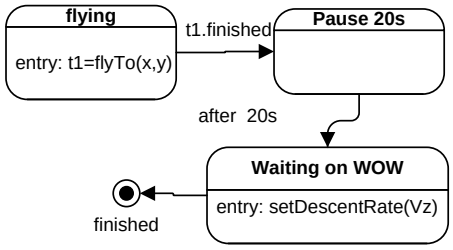

Figure 7: Procedure and Generated State Semantics for a `landAt` Procedure

may be described using state machines as illustrated in Figure 7. The *r* compiler automatically generates these state machines.

4. Discrete States

Agents themselves can have explicitly specified finite state machine behavior already alluded to in Figure 5 and earlier text. Transition between states is commanded by the `goto` construct whose syntax is given below.

```
goto <targetstate> [if (<expression>)];
```

The `goto` construct may be issued by a task or may be issued by the state itself when a guard condition (`if`) becomes true. State semantics become useful in dealing with exception situations, where select nominal procedures may be overridden to behave differently.

Listing 2: Discrete States Example

```
class FMS {
    Aircraft ac = ref : #
    state inAir {
        goto onGround if (ac.wow);
        state emergencyClimb {
            proc pullUp() {}
            ....
        }
        state cruise {
            proc land() { goto landing; }
            proc collisionWarning() { goto emergencyClimb; }
        }
        state landing {
            proc deployGear() {
                waitfor(openDoors()) {
                    lowerGear();
                }
            }
        }
        state climbToCruise() {....}
```

```
    }
    state onGround {
        goto inAir if(not ac.wow);
    }
}
```

IV. Conflict Resolution Example Script

```
class atc {
    proc resolveConflict(Aircraft ac1, Aircraft ac2) {
        int choice = 0.5 : # random choice of algorithm
        waitfor( areAircraftInConflict(ac1,ac2) ) {
            choice = random();
            if(choice < 0.9) {
                t = speedControl(ac1,ac2);
            } else {
                t = altitudeControl(ac1,ac2);
            }
            waitfor t.finished;
        }
    }

    proc altitudeControl(Aircraft ac1, Aircraft ac2) {
        double conflictTime = 0 : # time to conflict
        Aircraft higher     = 0 : # A/C that is higher in altitude
        Aircraft lower      = 0 : # A/C that is lower in altitude
        conflictTime = getConflictTimeAhead(ac1,ac2);
        waitfor( areAircraftInConflict(ac1,ac2)
                 and getConflictTimeAhead(ac1,ac2) < 240
                 and altSep(ac1,ac2) > 1000
                 and ac1.fpa < -0.5 and ac2.fpa < -0.5 ) {

            if (ac1.altitude > ac2.altitude) {
                higher = ac1;
                lower  = ac2;
            } else {
                higher = ac2;
                lower  = ac1;
            }

            if (conflictTime > 60) {
                t1 = changeAltWithVS(higher,targetAltitude,lower.VS);
                waitfor (180s) then {
                    higher.resumeCourse();
                    successResolution(ac1,ac2);
                }
            } else {
                t2 = changeAltWithVS(higher,higher.altitude,0.0);
                waitfor (60s) then {
                    t3 = changeAltWithVS(higher,targetAltitude,lower.VS);
                    waitfor (180s) then {
                        higher.resumeCourse();
                        successResolution(ac1,ac2);
                    }
```

```
                }
            }
        }
    }


    proc speedControl(Aircraft ac1, Aircraft Ac2) {
        waitfor(areACInConflict(ac1,ac2)) then {
            leader   = getLeadAC(ac1,ac2);
            trailing = getTrailAC(ac1,ac2);
            if(leader.changingSpeed and leader.commandedSpeed < leader.nextWayPointSpeed + 5) {
                t = leader.changeSpeed(leader.speed + 50);
                changed = leader;
            } elif(trailing.changingSpeed and trailing.commandedSpeed > trailing.nextWayPointSpeed
                t = trailing.changeSpeed(trailing.speed - 50);
                changed = trailing;
            }

            waitfor(120s) then {
                changed.resumeSpeed();
                successResolution(ac1,ac2);
            }
        }
    }
}
```

V. Conclusions

In many cases, research code is a conglomeration of C++ code, Matlab scripts, XML and Lisp. When integrated, a combination of tools may serve its purpose well, however, it is generally cumbersome and may not easily scale to larger problems or be flexible enough to accommodate new constructs.

The language described in this paper co-locates the declaration of continuous states, discrete states as well as procedures allowing a much simpler (and hence easily understandable) description of the problem and its solution. We found that most procedures may be represented as hierarchical-state-machines.[7] This is especially true when propositional and temporal logic is employed. Procedures are able to decide actions based on complex conditions that occur over time and are able to hold-off on decision making until information becomes available. By automatically generating the state-machine semantics needed for procedures, a higher-level description of solutions is possible. The explicit definition of states allows fine grained description of complex agent behavior when task semantics are not sufficient. The eventual translation to C++ using a translator (the *r* compiler) results in fast code capable of modeling dynamical systems as well as procedures.

References

[1]Kuchar, J. K. and Yang, L. C., "A Review of Conflict Detection and Resolution Modeling Methods," *IEEE Transactions On Intelligent Transportation Systems*, Vol. 1, No. 4, December 2000, pp. 179–189.

[2]Kim, S. Y., Lee, S. M., and Johnson, E. N., "Analysis of Dynamic Function Allocation between Human Operators and Automation Systems," *AIAA Guidance Navigation and Control Conference*, Honolulu, HI, August 2008.

[3]Freed, M., Dahlman, E., Dalal, M., Harris, R., and Jew, R., *The APEX Reference Manual*, NASA Ames, Ames Research Center, Moffet Field, California, 2nd ed., March 2004.

[4]Gat, E., "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents," *IEEE Aerospace Conference*, Aspen, CO, USA, February 1997.

[5]Simmons, R., "Structured Control for Autonomous Robots," *IEEE Transactions on Robotics and Automation*, Vol. 10, No. 1, 1994, pp. 34–43.

[6]Simmons, R. and Apfelbaum, D., "A Task Description Language for Robot Control," *Conference on Intelligent Robots and Systems*, Victoria, B.C, Canada, October 1998.

[7]Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, No. 3, 1987, pp. 231–274.

[8]Samek, M., *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, Elsevier Science & Technology Books, USA, January 2002.